SWT - CA2

فرشته باقری - محمد امانلو

۸۱۰۱۰۰۰۸۴ - ۸۱۰۱۰۰۰۸۹

https://github.com/MohammadAmanlou/SWT-Fall03

733bf11ba339b566992b019fbeb52dafe94f926e

سوال اول - انواع Verification

State Verification: در این نوع صحت‌سنجی، حالت انتهایی سیستم یا یک موجودیت را بعد از اجرای یک متد تست می‌کنیم. در این نوع از صحت‌سنجی تمرکز بر روی نتیجه اجرای یک متد (یا دنباله ای از آنها) است و اینکه چگونه به آن state رسیده‌ایم، اهمیتی ندارد. برای مثال برای تست کردن یک متد که یه یک user را اضافه می‌کند، صرفا چک می‌کنیم که در انتها کاربر مدنظر به سیستم اضافه شده باشد.
در این نوع تست‌ها، با استفاده از stub می‌توانیم ورودی‌های سیستم را کنترل کنیم و وضعیت نهایی را بدون تمرکز بر روی مراحل میانی یا فراخوانی متدها بررسی کنیم.

Behavior Verification: در این نوع تست‌ها، چک می‌کنیم که چگونه به این state از برنامه رسیده‌ایم. برای مثال در این روش بررسی می‌کنیم که یک متد خاص چند بار و با چه آرگومان‌هایی فراخوانی شده است. در مثال اضافه کردن یک کاربر، در این تست ما چک می‌کنیم که متد ما یک بار با آرگومان های درست صدا زده شده باشد.
در این نوع صحت‌سنجی، با استفاده از mock ها می‌توانیم interaction های متد/ کلاس مدنظر را با dependency های خود بررسی کنیم و چک کنیم که دقیقا رفتار مورد انتظار را انجام داده‌اند.

سوال دوم - Test Spy

در دنیای توسعه نرم‌افزار، تست واحد یا Unit Testing نقش بسیار مهمی در اطمینان از کیفیت و صحت عملکرد کد دارد. اما گاهی اوقات لازم است بخش‌هایی از کد را به صورت جداگانه و بدون تأثیر وابستگی‌های خارجی تست کنیم. در چنین شرایطی، مفهوم Test Spy یا جاسوس تست به کار می‌آید.
Test Spy در واقع نوعی از Test Double است که در تست‌های واحد استفاده می‌شود. وظیفه اصلی آن شبیه‌سازی رفتار یک شیء واقعی است، در حالی که اطلاعات مربوط به تعاملات و فراخوانی‌های انجام‌شده روی آن را نیز ضبط می‌کند. به عبارت ساده‌تر، Test Spy مثل یک ناظر عمل می‌کند که تمام فراخوانی‌ها و پارامترهای استفاده‌شده را ثبت می‌کند. این ابزار به ما کمک می‌کند تا نه‌تنها خروجی‌ها را بررسی کنیم، بلکه نحوه تعاملات داخلی و فراخوانی متدها را نیز نظارت کنیم.

اما چرا از Test Spy استفاده می‌کنیم؟ اول اینکه با استفاده از آن می‌توانیم اطمینان حاصل کنیم که متدهای مورد نظر با پارامترهای صحیح و در زمان مناسب فراخوانی شده‌اند. دوم، با حذف وابستگی‌های خارجی، می‌توانیم روی بخش خاصی از کد تمرکز کرده و تست‌های دقیق‌تری انجام دهیم. همچنین، با ضبط دقیق تعاملات، دقت تست‌ها افزایش می‌یابد و مطمئن می‌شویم که کد ما به درستی کار می‌کند. از طرفی، استفاده از Test Spy باعث کاهش پیچیدگی تست‌ها می‌شود؛ چون به جای نوشتن کدهای پیچیده برای شبیه‌سازی رفتار اشیاء، می‌توانیم فرآیند تست را ساده‌تر کنیم.

انواع Test Spy بر اساس نحوه پیاده‌سازی و کاربردشان به سه دسته تقسیم می‌شوند: Spyهای دستی، Spyهای خودکار با استفاده از کتابخانه‌ها و Spyهای ترکیبی.

Spyهای دستی به این صورت هستند که برنامه‌نویس خودش به صورت دستی یک کلاس یا شیء ایجاد می‌کند که رفتار مورد نظر را شبیه‌سازی کرده و اطلاعات لازم را ذخیره می‌کند. مزیت این روش این است که کنترل کامل بر روی رفتار و داده‌های ضبط‌شده داریم و انعطاف‌پذیری بالایی در پیاده‌سازی وجود دارد. اما از طرف دیگر، نیاز به کدنویسی اضافی و زمان‌بر است و ممکن است خطاهای انسانی رخ دهد. مثلاً فرض کنید متدی داریم که باید یک ایمیل ارسال کند؛ می‌توانیم یک Spy دستی ایجاد کنیم که هر بار متد ارسال ایمیل فراخوانی می‌شود، تعداد دفعات فراخوانی و پارامترهای آن را ثبت کند.

Spyهای خودکار با استفاده از کتابخانه‌ها از ابزارها و کتابخانه‌های موجود برای ایجاد Spy به صورت خودکار استفاده می‌کنند. کتابخانه‌هایی مثل Mockito در جاوا، Sinon.js در جاوااسکریپت یا unittest.mock در پایتون از این دسته هستند. مزیت این روش این است که زمان و تلاش مورد نیاز برای نوشتن تست‌ها کاهش می‌یابد، امکانات پیشرفته‌ای برای نظارت بر تعاملات در اختیار داریم و احتمال خطاهای انسانی کمتر می‌شود. اما نیاز به یادگیری و استفاده از کتابخانه‌های اضافی داریم و شاید در برخی موارد انعطاف‌پذیری کمتری نسبت به Spyهای دستی داشته باشد. مثلاً در جاوا با استفاده از Mockito می‌توانیم یک Spy از یک ArrayList ایجاد کنیم و تعاملات با آن را نظارت کنیم:

```
List<String> spyList = Mockito.spy(new ArrayList<>());
spyList.add("Test");
Mockito.verify(spyList).add("Test");
```

Spyهای ترکیبی ترکیبی از دو روش بالا هستند و ممکن است در پروژه‌های بزرگ و پیچیده استفاده شوند. این روش مزایای هر دو روش را دارد و انعطاف‌پذیری بالایی ارائه می‌دهد، اما پیاده‌سازی و نگهداری آن پیچیده‌تر است و نیاز به تجربه و دانش بیشتری دارد. برای مثال، در یک پروژه بزرگ ممکن است از Spyهای خودکار برای بخش‌های عمومی استفاده کنیم و برای موارد خاص که نیاز به کنترل بیشتری دارند، Spyهای دستی را به کار ببریم.

به طور کلی، استفاده از Test Spy به ما کمک می‌کند تا تست‌های واحد قوی‌تر و دقیق‌تری بنویسیم و مطمئن شویم که کد ما به درستی کار می‌کند. با نظارت بر تعاملات داخلی و فراخوانی متدها، می‌توانیم اطمینان حاصل کنیم که همه چیز مطابق با انتظارات ما پیش می‌رود و در نتیجه نرم‌افزاری با کیفیت بالاتر تولید کنیم.
به طور خلاصه:
- Test Spy نوعی Test Double است که تعاملات و فراخوانی‌های انجام‌شده را ضبط می‌کند.
- به ما کمک می‌کند تا فراخوانی متدها و پارامترهای آن‌ها را نظارت کنیم.
- سه نوع اصلی دارد: Spyهای دستی، خودکار و ترکیبی.
- انتخاب نوع مناسب Spy به نیازمندی‌ها و پیچیدگی پروژه بستگی دارد.
و در نهایت فایده اش:
با فهم بهتر از Test Spyها و نحوه استفاده از آن‌ها، می‌توانیم مشکلات پنهان در کد را پیش از انتشار شناسایی و برطرف کنیم که باعث بهبود تجربه کاربری و اعتماد به نرم‌افزار می شود و در نهایت منجر به تولید نرم‌افزاری با کیفیت بالاتر و کاهش خطا ها در محیط تولید خواهد شد.

سوال سوم - Shared Fixture

الف) زمانی که setUp بسیار پیچیده باشد و منابع زیادی را استفاده کند یا به منابع خارجی مانند دیتابیس یا سرور نیاز داشته باشد، استفاده از fresh fixture می‌تواند هزینه‌بر و زمان‌بر شود. در این مواقع، به شرطی که تست‌ها بتوانند به صورت امن از یک fixture اشتراکی کنند به طوری که روی نتایج همدیگر اثر نگذارند و ترتیب اجرای تست‌ها منجر به خروجی متفاوت نشود، بهتر است از یک shared fixture استفاده کرد. برای مثال اگر همه تست‌ها فقط از fixture داده می‌خوانند و آن را تغییر نمی‌دهند، استفاده از shared fixture مناسب‌تر است.

ب) **Lazy Setup:** در این نوع از setup، منابع فقط در صورتی initialize می‌شوند که یک تست به آنها نیاز داشته باشد که باعث کاهش زمان setup های غیر ضروری می‌شود. همچنین باعث می‌شود منابعی که به ندرت استفاده می‌شوند همواره فضای memory را اشغال نکنند مگر اینکه نیاز به دسترسی به آنها باشد. اما زمانی که تعداد زیادی از تست‌ها به یک setup که پیچیده و resource-intensive است، نیاز دارند، استفاده از این روش باعث افزایش زمان اجرای تست‌ها خواهد شد. از طرفی اگر این setup ها حالت fixture را تغییر دهند، ممکن است انجام چندباره آن در تست های مختلف منجر به inconsistent state شود. اگر تست‌های ما مقدار زیادی منابع مشترک استفاده کنند، بهره‌گیری از این روش باعث افزایش duplication نیز می‌شود.
Suite Fixture Setup: اگر دسته بزرگی از تست‌ها به یک منبع یکسان نیاز دارند، استفاده از این روش می‌تواند هزینه و زمان اجرای تست‌ها را کاهش دهد. همچنین با این روش به مشکل حالت inconsistent برنخواهیم خورد. از معایب این روش می توان به این اشاره کرد که اگر تستی حالت fixture را تغییر دهد، ممکن است نتایج

تست‌های دیگر را تحت تاثیر قرار دهد. همچنین چون Setup برای کل تست‌های suite انجام شده است، در طی کل زمان اجرا منابع سیستم را استفاده می‌کند حتی اگر تست در حال اجرا به آن احتیاجی نداشته باشد. این روش flexibility کمتری دارد و اگر تست‌های متفاوت شرایط اولیه کمی مختلفی را نیاز داشته باشند، نمی‌توان به سادگی از آن استفاده کرد.

ج) اگر استفاده از fresh fixture ها ممکن نیست یا بسیار هزینه‌بر است(مثل حالت‌هایی که دیتابیس داریم)، می‌توان از مکانیزم های transaction rollback استفاده کرد تا بعد از هر تست، fixture به حالت اولیه خود برگردد. همچنین می‌توان fixture را صرفا به صورت read-only تعریف کرد تا تست‌ها نتوانند آن را تغییر دهند، در این صورت استفاده اشتراکی از آن منجر به مشکلی نخواهد شد. همچنین در برخی مواقع می‌توان از mock یا stub استفاده کرد. این روش‌ها به ما اجازه می‌دهند که interaction یک متد/ یونیت را با fixture بدون نیاز به تغییر آن شبیه‌سازی کرد.